Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation

Chi-Keung Luk
Department of Computer Science
University of Toronto
Toronto, Canada M5S 3G4
luk@eecg.toronto.edu

Todd C. Mowry
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213
tcm@cs.cmu.edu

Abstract

By optimizing data layout at run-time, we can potentially enhance the performance of caches by actively creating spatial locality, facilitating prefetching, and avoiding cache conflicts and false sharing. Unfortunately, it is extremely difficult to guarantee that such optimizations are safe *in practice on today's machines, since accurately updating* all *pointers to an object requires perfect alias information, which is well beyond the scope of the compiler for languages such as C. To overcome this limitation, we propose a technique called* memory forwarding *which effectively adds a new layer of indirection within the memory system whenever necessary to guarantee that data relocation is always safe. Because actual forwarding rarely occurs (it exists as a safety net), the mechanism can be implemented as an exception in modern superscalar processors. Our experimental results demonstrate that the aggressive layout optimizations enabled by memory forwarding can result in significant speedups—more than twofold in some cases—by reducing the number of cache misses, improving the effectiveness of prefetching, and conserving memory bandwidth.*

1. Introduction

As the gap between processor and memory speeds continues to grow, memory latency is becoming a key performance bottleneck. In addition, there is growing concern that the *bandwidth* to the off-chip memory hierarchy may also limit performance [5]. The primary mechanism for reducing both the off-chip latency and bandwidth requirements is the on-chip cache hierarchy. While caches are an important step toward addressing these problems, they face several well-known limitations. For example, multi-word cache lines can improve performance by prefetching useful data when spatial locality is abundant, but they can also waste bandwidth and displace useful data when it is not. Caches can suffer pathologically bad behavior due to either mapping conflicts or *false sharing* [21, 34]. Prefetching techniques [9, 25, 31] can potentially hide cache miss latency, but only if they can predict access patterns ahead of time, and only if there is sufficient memory bandwidth.

A brute force approach of simply making caches larger is not likely to solve these problems, in part because application problem sizes are also increasing rapidly, and also because cache sizes are constrained by the requirement of low access latency and by hardware resource limitations. In addition, recent studies [5, 18] have shown that the effectiveness of caches is often low because a significant fraction of cached data is not reused before it is displaced from the cache. Hence the first problem to address is managing existing caches more *intelligently* to improve their effectiveness.

Cache performance depends on two factors: *when* data items are accessed, and *where* they exist in the address space. Therefore, software-based techniques for improving cache performance typically do one of two things: they either restructure the *computation*, or else they restructure the *data layout*. The idea behind restructuring the computation is that given a fixed data layout, we would like to manipulate the ordering of accesses such that multiple accesses to the same data item (or cache line) occur close together in time, thereby enhancing locality [8, 37]. In contrast, the idea behind optimizing the data layout is that given that a set of data items are accessed close together in time in the original computation, we would like to actively arrange them in the address space such that: (i) we *create spatial locality* by allocating them at contiguous addresses (thereby enhancing the effectiveness of long cache lines and simplifying prefetch address generation); (ii) we *avoid cache conflicts* by ensuring that they do not reside in separate lines which map into the same cache sets; and (iii) we *avoid false sharing* by ensuring that items accessed by different processors fall within separate cache lines. While both approaches have received considerable attention in the past, our focus in this study is on facilitating data layout optimizations.

There are two possibilities for when we manipulate data layout. The first approach—which we call *static placement*—is to assign an object to its optimized address when it is created [6]. The second approach is to move an object (perhaps more than once) *after* it has been allocated; we refer to this latter approach as *data relocation* (or simply *relocation*). The advantage of static placement is its simplicity. The advantage of relocation, however, is that it can adapt to dynamic program behavior. Previous studies have shown that relocation-based optimizations such as *copying* [23, 33] and *clustering* [11] can offer impressive performance gains.

In general, relocation-based data layout optimizations involve the following three steps:

1. **Guaranteeing Correctness:** Either the programmer or the compiler must prove that relocating the data will *never* break the program; otherwise, the optimization is unsafe.
2. **Estimating the Cost/Benefit Tradeoff:** The potential optimization should only be performed if the performance benefit is expected to outweigh the overheads involved in relocating the data. This estimation could be based on some combination of programmer knowledge, static compiler analysis, profiling feedback, or run-time information.
3. **Generating Relocation Code:** Additional code must be inserted to perform the actual data relocation at run-time.

Despite the high performance potential of many relocation-based optimizations, the key stumbling block which often prevents them from being used in practice is the first step—i.e. *guaranteeing correctness*. To safely move data, we must guarantee that any future references to the object will find it at its new location. The fundamental problem is that updating the precise set of pointers[1] to a given object requires perfect aliasing information related to that object. In general, computing such precise information is beyond the capabilities of the compiler,[2] and is even quite difficult for the programmer for large programs. In the face of uncertainty, we must conservatively assume that relocating an object will break the program—no matter how unlikely this may seem in reality—and therefore the optimization cannot be performed.

There is one mechanism in modern systems which provides a very limited form of safe data relocation: the virtual memory system. The operating system can relocate an entire page of memory in the physical address space without breaking the program by simply copying the page and updating its virtual-to-physical mapping. One cache optimization which exploits this flexibility is *page coloring* [22], whereby the operating system attempts to avoid mapping conflicts in large off-chip caches. Therefore, by adding a layer of indirection *within* the memory system, we can move data safely and transparently without any special language or compiler support. Unfortunately, the virtual memory system only provides this flexibility at the granularity of an entire page. To actively create spatial locality *within* a cache line, we must have this flexibility at a *word granularity*. However, applying standard virtual memory techniques at such a fine granularity—i.e. setting the page size to be one word—is not a viable solution, due to the enormous overheads that this would involve. (Not only would the number of page table entries and the TLB size grow enormously, but also the cache tags would have to be maintained at a word granularity.) Instead, we propose a completely different solution.

1.1. Our Solution: Memory Forwarding

To give software the flexibility to apply relocation-based data layout optimizations at any time without concern over violating program correctness, we propose a mechanism called *memory forwarding* which guarantees the safety of relocation at a word granularity. (In our discussion, we define the "word" size to be equal to the size of a pointer.) The basic idea behind memory forwarding is that when we relocate an object, we store its new address in its old location and mark the old location so that hardware recognizes it as a *forwarding address*. Therefore if the program accidentally accesses the old address, the hardware will automatically forward the reference to the object's new location, thereby always guaranteeing the correct result. Moreover, our scheme only pays the run-time overhead of an extra indirection when it is actually necessary—i.e. when the alternative is to violate program semantics. In the far more common cases of references to non-relocated objects, or references that have been properly updated to point to the new addresses of relocated objects, our scheme imposes no performance overhead. The space overhead of our scheme is also low (a 1.5% fixed memory cost on a 64-bit architecture).

With memory forwarding support, the decision of whether to apply a relocation-based optimization reduces solely to evaluating its cost/benefit performance tradeoffs. In effect, memory forwarding enables software to optimistically *speculate* that when it relocates an object, it has successfully updated all pointers to that object to point to its new location. If the speculation fails, then there is a recovery cost (i.e. dereferencing the forwarding address), but the execution still proceeds correctly. Therefore, as in all forms of speculation, one is gambling that the speculation is correct often enough that the benefit outweighs the cost. Another feature of our mechanism which helps improve these odds is that software can optionally specify that dereferencing a forwarding address will invoke a user-level trap that enables software to update the offending pointer to point to the object's new address. Hence software can learn from its mistakes to avoid repeating them.

1.2. Related Work

It is interesting to note that the work which is most closely related to our study occurred well over a decade ago in the context of architectures that directly supported the *Lisp* programming environment [29, 32]. Performance concerns were quite different back then: main memory was relatively small and expensive, and cache miss latencies were less problematic because the gap between microprocessor and memory speeds was dramatically smaller. (In fact, a number of microprocessor-based systems did not even have caches.) Therefore the primary concern in optimizing memory performance back then was minimizing the overall *space* requirements of a program, so that it would fit into main memory and avoid paging to disk. Two aspects of the Lisp environment made this challenging: the need to perform automatic garbage collection, and the relative space inefficiency of the ubiquitous list structures in the language. In addition, another aspect of the Lisp language which resulted in specialized hardware support was the need to determine object data types at run-time.

Although the performance goals which inspired specialized hardware and software support in these Lisp machines are quite different from our goal of improving cache performance, there are nonetheless a number of interesting overlaps between our support and some features of these earlier machines. We now discuss the connections between this previous work and our study from three different perspectives: tagged memory, garbage collection, and data layout optimizations.

Tagged Memory: To make objects self-descriptive with respect to their types, a number of Lisp architectures [29, 32] associated a *tag* with each memory location. As we will see later in Section 2.1, our memory forwarding scheme also requires a form of tagged memory to distinguish forwarding addresses from normal data. A key difference, however, is that the tags in Lisp machines provided much more functionality than in our case, and therefore they required more overhead. For example, the SPUR architecture [32] added eight bits of tag to each 32 bits of memory (a 25% overhead), whereas our scheme only requires one tag bit per 64 bits of memory (a 1.5% overhead) in a modern 64-bit architecture.

A fact that is even more relevant to our study is that a form of memory forwarding (using tagged memory) has appeared in previous Lisp machines, albeit for a very different purpose. The concept of an *invisible pointer* (which is similar to our forwarding address) was proposed twenty-four years ago by Greenblatt [15], and the Symbolics 3600 [29] used one of its tags to implement

[1] We use the term "pointers" loosely to refer to any mechanism for generating an address pointing to the object in question.

[2] This is especially true for heap-allocated objects in languages like C.

a *forwarding pointer*. The motivation behind these mechanisms was threefold: to enable the insertion of an item into a *cdr-coded* list [2], to facilitate incremental garbage collection, and to implement overlapping arrays. In contrast, our focus is on improving the cache performance of programs written in C, and therefore none of these issues apply. In essence, what we are doing is taking a very old mechanism and adapting it to a completely new purpose within the context of modern out-of-order superscalar processors.

Garbage Collection: A common feature among these Lisp machines is that they support some form of automatic garbage collection. Garbage collection algorithms involve phases where they identify two classes of data items: those that can be *reclaimed*, and those that can be *relocated*. A data item can be reclaimed when it can no longer be accessed through any pointers that are still active, and a data item can be relocated if *all* pointers to the old location can be updated to point to the new location. In both cases, the key challenge is identifying all pointers which point to the given location. In languages such as Lisp, ML, and Java, where the use of pointers is either restricted or disallowed altogether, one can solve this problem in practice. In contrast, in languages such as C and C++ which do not restrict pointer usage, one generally cannot determine which pointers point to a given object, and therefore automatic garbage collection (and data relocation) is extremely difficult. Finally, it is interesting to note that a form of memory forwarding is used in *copying garbage collectors* [10, 28], whereby the forwarding addresses are used to preserve data consistency during the distinct phases when collection takes place.

Data Layout Optimizations: An important topic in Lisp research is how to represent list structures compactly. List compaction can be performed either separately or during garbage collection. Most of the list compaction techniques designed for Lisp [1, 2, 16] involve either moving or copying the original list to a new, denser set of locations. As we discussed above, data relocation in Lisp does not pose the safety problems that we encounter in C. However, our memory forwarding support gives us the flexibility to exploit some of these same list compaction techniques—e.g., a technique called *list linearization* [13]—for the sake of improving spatial locality in C programs.

1.3. Objectives of This Study

This paper makes the following contributions. First, we propose a solution to the problem of safely relocating data at a fine granularity to improve the cache performance of programs written in languages such as C which do not support garbage collection. Although the concept of memory forwarding was proposed over two decades ago in the context of Lisp machines, to the best of our knowledge, we are the first to propose that it be adapted to facilitate a broad class of data layout optimizations to improve cache performance. Second, we discuss how memory forwarding can be implemented within modern out-of-order superscalar processors (which are quite different from the processors in which other forms of forwarding have been implemented in the past). Third, we suggest a number of optimizations which can benefit from memory forwarding. Finally, we quantitatively evaluate the benefits and overheads of our scheme by using it to apply a number of different run-time locality optimizations to a collection of non-numeric applications running on a modern superscalar processor.

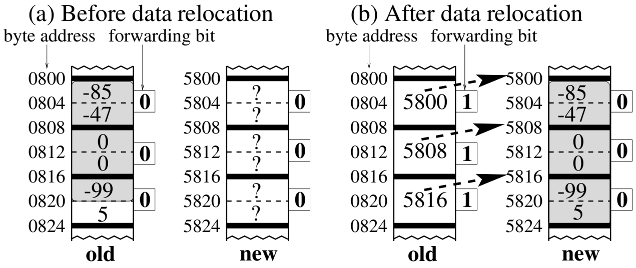

Figure 1. Example of data relocation with memory forwarding. (A memory word is 8 bytes, and addresses are in decimal.)

The remainder of this paper is organized as follows. We begin in Section 2 with an overview of memory forwarding and how it can be used. Section 3 discusses issues related to implementing memory forwarding in a modern processor. Sections 4 and 5 present our experimental methodology and experimental results, respectively, to demonstrate the usefulness of the mechanism. Finally, we conclude in Section 6.

2. Memory Forwarding

We now discuss the basic concepts behind memory forwarding, a number of applications of this mechanism, and some issues related to its performance.

2.1. Basic Concepts

Memory forwarding enables *aggressive* yet *safe* data relocation. As we mentioned earlier in Section 1.1, the basic idea is to store the new address of an object into its old memory location, and to mark this old location as a *forwarding address*. Whenever a forwarding address is accessed, the hardware will automatically dereference that location to find the object at its new location.

There are three implications of this mechanism in terms of memory storage. First, the minimum unit of data that can be relocated is the width of a pointer—which we refer to as a "word" throughout this paper—since otherwise there would not be enough space to store the forwarding address.[3] Note that it is possible to relocate byte-sized objects—this simply means that enough neighboring bytes must be moved at the same time to comprise an entire word. Second, a chunk of data that is relocated must be word-aligned, so that the alignment of the forwarding address is predetermined. Note that this still allows us to perform byte-sized loads and stores to forwarded objects—the byte offset into the new location is simply assumed to be the same as it was at its original address. We consider these first two restrictions to be quite minor, especially given that our only option for safe relocation today is page-sized, page-aligned chunks of data. Finally, to enable the hardware to distinguish forwarding addresses from regular data, we attach a one-bit tag (called a *forwarding bit*) to each word in memory. For a 64-bit architecture, this results in a space overhead of only 1.5%, and therefore is reasonably efficient.

Figure 1 shows a simple example of how the memory contents and forwarding bits are modified upon data relocation. Assume

[3] One could imagine creating a more elaborate scheme for compressing the size of forwarding address pointers (e.g., by restricting the distance between the old and new address to something that fit within its former size), but this would involve additional complexity and fancier tag storage, so we do not consider such an approach further.

that we have a 64-bit architecture, and that we would like to relocate five 32-bit elements from addresses `0800-0816` to addresses `5800-5816` (these addresses are in decimal notation). Figure 1(a) shows the memory contents and forwarding bits before relocation (note that none of the forwarding bits have been set). To relocate a word, we first copy it to its new location, and then we *simultaneously* write its new address into its old location and set the corresponding forwarding bit at the same time. Figure 1(b) shows the state of memory after the relocation. Notice that to relocate the 32-bit subword at address `0816`, we must also relocate the 32-bit subword at address `0824` (which contains the value 5) along with it. After the relocation, a 32-bit load of the subword at address `0804` will be forwarded to address `5804`—which is computed by adding the forwarding address (`5800`) to the byte offset within the word (`4`)—thereby returning the correct value of `-47`.

To simplify our discussion throughout the remainder of the paper, we now define two terms which we will use frequently:

Initial address: The address of the *first* location accessed by a memory reference. For example, in Figure 1(b), the initial address of a write to word `0816` is `0816` itself.

Final address: The address of the *last* location accessed by a memory reference. For example, in Figure 1(b), the final address of a write to word `0816` is `5816`. When data is not forwarded, the final address equals the initial address.

In addition to preserving the correctness of pointer *dereferences*, another concern with data relocation is preserving the correctness of pointer *comparisons*. In the presence of memory forwarding, it is possible that two pointers with distinct *initial* addresses may in fact point to the same object (i.e. share the same *final* address). Hence to preserve correctness, explicit pointer comparisons[4] should be performed with respect to their *final* addresses. Although our memory forwarding hardware does not perform this check automatically, the compiler can easily insert additional instructions (described later in Section 3) to look up the final addresses for these comparisons. We implemented such a compiler pass, and the resulting software overhead is included in our performance results. As we will see later in Section 5, this overhead does not present a problem.

2.2. Applications of Memory Forwarding

While the act of dereferencing a forwarding address clearly does not improve performance on its own, the advantage of memory forwarding support is that it enables a wide range of data layout optimizations which can enhance cache performance. Not only are these optimizations useful for mitigating the impact of memory latency, they can also be used to conserve memory *bandwidth*. We now briefly describe some of these potential optimizations.

Improving Spatial Locality: A straightforward method of actively improving spatial locality is to take data items which are accessed close together in time, but which are scattered sparsely throughout the address space, and pack them into adjacent memory locations. This form of data packing makes cache lines much more effective, and it can potentially reduce the number of capacity, compulsory, and conflict misses. Not only does this improve performance, it can also reduce memory bandwidth consumption, which in turn can help reduce power consumption (which is becoming an increasingly important concern).

[4] In reality, we only need to worry about cases where the pointers potentially point to the same type of relocatable object.

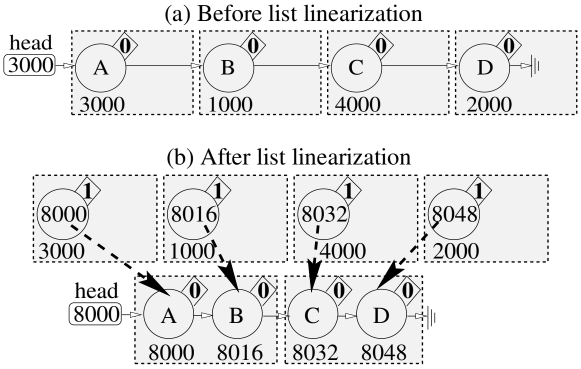

Figure 2. Example of list linearization with memory forwarding, assuming that cache lines and list elements are 32 and 16 bytes long, respectively. (Addresses are in decimal.)

A good example of a technique which uses packing to enhance spatial locality is *list linearization*. As we will see later in Section 5, this technique can offer dramatic performance improvements. List linearization has been an important technique for compacting lists in Lisp programs, and can eliminate as much as half of the space consumption [13]. The idea behind list linearization is to relocate the nodes of a linked list so that they reside in contiguous memory locations. Depending on whether the list structure continues to change over time, the linearization process can be invoked either just once, or else periodically to adapt to the changing structure. Although list linearization can potentially offer large performance gains, it is very difficult to safely use this optimization in practice for general C programs due to the possibility of pointers outside of the linked list itself pointing to list elements. Fortunately, with memory forwarding support, we can apply list linearization at any time without worrying about whether all potential pointers to list elements have been properly updated.

Figure 2 shows an example of list linearization with memory forwarding. Before linearization, the four nodes of the list (i.e. nodes `A`, `B`, `C`, and `D`) are scattered throughout memory such that they reside in four separate cache lines, as shown in Figure 2(a). List linearization packs the four nodes into a contiguous memory region starting at location `8000`, as shown in Figure 2(b). As a result, the four relocated nodes occupy only two cache lines, rather than four, thereby potentially eliminating half of the cache misses due to this list as we continue to revisit it. Note that the forwarding addresses and forwarding bits have been set properly such that we will still maintain correct execution even if a stray pointer accesses a list element at its old address. However, we expect that most accesses to the list will find it directly at its new address, thereby enjoying the enhanced spatial locality.

Increasing Prefetching Effectiveness: The effectiveness of prefetching for *non-numeric* applications is largely limited by the difficulties in generating prefetch addresses early enough [25]. For example, consider the linked list in Figure 2(a), and assume that we need to prefetch three nodes ahead to hide the entire miss latency. Therefore, we would want to prefetch node `D` as soon as we arrive at node `A`. However, the problem is that we do not know the address of node D until we have dereferenced nodes A, B, and C—this is known as the *pointer-chasing problem* [25]. In contrast, after the list is linearized, we can trivially prefetch node D at node A by simply prefetching the next cache line, thus avoiding any pointer chasing. (We referred to this technique as *data linearization prefetching* in an earlier publication [25].)

Reducing Cache Conflicts: *Data copying* [23] was originally proposed to reduce conflict misses within tiled (or "blocked") numeric applications. Since a given tile is reused many times after it is brought into the cache, it is particularly problematic if different elements *within* the tile conflict with each other. To avoid this problem, the data copying optimization first copies a tile to a contiguous set of addresses in a temporary array before using it; since these locations do not conflict with one another, the problem is eliminated. Another technique called *data coloring* [11] was proposed as a method of reducing conflict misses in pointer-based data structures. The idea is to partition the cache into logically separate regions (or *colors*). By relocating data structure elements which are accessed close together in time to separate regions of the cache, conflict misses can be avoided. Memory forwarding can help facilitate both copying and coloring techniques by guaranteeing that they are safe.

Reducing False Sharing: In cache-coherent shared-memory multiprocessing systems, *false sharing* [21, 34] occurs when two or more processors access distinct data items which happen to fall within the same cache line (which is the unit of coherence), and at least one access is a write. False sharing can hurt performance dramatically as the line ping-pongs between processors despite the fact that no real communication is taking place. By relocating those unrelated data items to distinct cache lines, false sharing can be avoided. Memory forwarding would be especially helpful in avoiding false sharing in *irregular* shared-memory applications, where proving that data items can be safely relocated is difficult.

In summary, memory forwarding enables a broad range of relocation-based optimizations; we have presented just a partial list of such optimizations. We would also like to emphasize that these optimizations are applicable not only to caches but also to the other levels of the memory hierarchy. For example, we can apply data relocation to improve the spatial locality within pages (and hence on disk) for out-of-core applications.

2.3. Performance Issues

A relocation-based optimization will improve overall performance if two conditions hold: (i) the new data layout actually provides better memory performance than the original layout; and (ii) the gain in the memory performance outweighs the optimization overhead. This overhead includes the extra execution time due to actually relocating the data, and may also include forwarding overhead if any references actually need to be forwarded after the relocation. While the overhead of relocating the data may seem to be a concern at first glance, our experimental results indicate that it is usually not a problem because relocation is invoked infrequently and modern processors can execute multiple instructions per cycle. In addition, we find that the performance overhead of forwarding is negligible in many cases because most data references are updated properly and do not need to be forwarded. We observe that the real performance concern is ensuring that the reorganized data layout actually delivers higher memory performance than the original layout.

R = Read_FBit(word* A)**:** Read the forwarding bit of the word at address A into register R.

R = Unforwarded_Read(word* A)**:** Read the value stored in the word at address A into register R, with forwarding disabled. If the word's forwarding bit is set, this is a forwarding address; otherwise this is a regular data value.

*A = Unforwarded_Write(register word R**,** bit B)**:**[a] Write the value of register R into the word at address A *and* set the word's forwarding bit to B *atomically*, with forwarding disabled.

[a] If an instruction of the underlying ISA cannot have three operands, we can have two separate instructions for Unforwarded_Write(R,0) and Unforwarded_Write(R,1).

Figure 3. Proposed instruction set extensions to support memory forwarding. (C syntax is used to improve readability.)

3. Implementation Issues

We now discuss the support that we need from the instruction set, the hardware, and the software to implement memory forwarding in modern superscalar processors.

3.1. Extensions to the Instruction Set Architecture

To exploit memory forwarding, the machine must have some way to manipulate the forwarding information—i.e. the forwarding addresses and the forwarding bits. Rather than taking a purely hardware-based approach, we propose to extend the underlying instruction set architecture (ISA) by adding a few instructions which will allow software to manipulate the forwarding information directly. The advantages of this approach are its programmability and flexibility. In addition, we expect the software overhead to be low since forwarding information changes relatively infrequently.

Figure 3 shows our proposed ISA extensions, which consist of three new instructions. Read_FBit allows software to check whether a given location contains a forwarding address or actual data. Unforwarded_Read and Unforwarded_Write allow software to manipulate memory with the forwarding mechanism disabled. For example, in Figure 1(b), a normal Read (i.e. with the forwarding mechanism enabled) of the word at address 0808 will get the forwarded value of 0, but an Unforwarded_Read of the same word will get 5808, which is the forwarding address. An Unforwarded_Write must change the word *and* its forwarding bit *atomically* in order to preserve data consistency.

To demonstrate how software can make use of these new instructions, Figure 4(a) shows two procedures for relocating a data object of size n_words from src to tgt, and then storing tgt as the forwarding address into src. Procedure Relocate() loops until a clear forwarding bit is read so that tgt will be appended at the end of the forwarding chain (if any). Figure 4(b) shows a procedure called ListLinearize() (which we will use frequently later in our experiments) which calls Relocate() to perform list linearization. The parameter head_handle is the address of the list head. Note that the *address* of the list head (rather than its value) is passed into ListLinearize() because we want

(a) Data Relocation

```
// src = address of the object before relocation
// tgt = address of the object after relocation
// n_words = number of words to relocate
void Relocate(word* src, word* tgt, int n_words) {
    boolean relocated;
    relocated = Read_FBit(src);
    while (relocated) {
        // loop until the final address is reached
        src = Unforwarded_Read(src);
        relocated = Read_FBit(src);
    }
    actualRelocate(src, tgt, n_words);
}

void actualRelocate(word* src, word* tgt, int n_words) {
    // relocate each word in the object
    for(; n_words > 0; --n_words) {
        word temp;
        // save the content of src
        temp = Unforwarded_Read(src);
        // setup the forwarding address and forwarding bit
        *src = Unforwarded_Write(tgt, 1);
        // copy the original content of src to tgt
        *tgt = Unforwarded_Write(temp, 0);
        // prepare for the next word
        src += 1; tgt += 1;
    }
}
```

(b) List Linearization

```
// a pool of space for data relocation
extern char* memory_pool;
// head_handle = address of the list head pointer
void ListLinearize(node ** head_handle) {
    node ** handle, *tgt;
    // start from the list head
    handle = head_handle;
    while (*handle) {
        // grab space from the pool
        tgt = (node*)memory_pool;
        // increment the pool pointer
        memory_pool += sizeof(node);
        // relocate the node pointed-to by handle to the address stored in tgt
        Relocate(*handle, tgt, sizeof(node)/sizeof(word));
        // append the relocated node to the linearized list
        *handle = tgt;
        // prepare for next node
        handle = &(tgt→next);
    }
}
```

Figure 4. Procedures using the proposed ISA extensions to implement (a) data relocation and (b) list linearization.

to modify the list head to point to the new locations after the relocation is performed. (This effect was illustrated earlier in Figure 2(b), where the value of `head` is changed to `8000` after the linearization.) By doing so, the next time that the list is accessed via the list head, the new locations will be accessed directly without touching the old locations. Finally, note that in Figure 4(b), the new locations for the relocated nodes are allocated from a pool of contiguous memory, thereby creating spatial locality.

3.2. Hardware Support

We now discuss the hardware modifications necessary to support memory forwarding. The key insight which helps us keep the hardware simple is that references which actually require forwarding are expected to occur *rarely* (if ever). The forwarding mechanism is simply a safety net which allows us to continue to preserve program correctness in case the unexpected happens. In other words, we can treat forwarding as an *exception*. We will design the hardware to be fast in the common case—i.e. a normal, non-forwarded reference—and we are less concerned about the performance penalty when forwarding is actually invoked, since that is rare. Hence a legitimate option is to use a processor's normal exception handling mechanism to implement forwarding.

One hardware requirement that was mentioned earlier in Section 2.1 is that we need tagged memory. A number of systems which supported tagged memory have been built in the past [29, 32]. One difference with our scheme (as discussed earlier) is that we require less tag storage overhead than previous schemes; otherwise, it is quite similar. We now discuss the more novel features of our hardware support in greater detail.

Dereferencing Forwarding Addresses: In the presence of memory forwarding, the data referencing mechanism must be able to follow *forwarding chains* of arbitrary lengths. More specifically, when a memory word is accessed by a data reference, its forwarding bit is tested. If this bit is set, then the original data address will be replaced by the contents of the word just accessed (which contains a forwarding address), and a new memory access using the forwarding address will be launched. This process repeats until a clear forwarding bit is read (we will discuss how cycles might be handled later in Section 3.2), at which point the data reference can proceed as usual. One option is to implement this dereferencing mechanism purely within hardware; another is to implement it using a software-based exception handler (where the exception is triggered by accessing a word with its forwarding bit set). With the ISA extensions that we propose, it would be straightforward for a software handler to chase the forwarding pointer chain. Although the forwarding bit cannot be tested until the memory location is brought into the primary cache, this is no different from the delays associated with checking ECC or parity bits.

Data Dependence Speculation: One consequence of memory forwarding is that we do not know the final data address of a given reference until the reference is nearly completed. This delayed generation of the final address poses a potential problem in out-of-order superscalar machines. These machines normally allow a load access to proceed before an earlier store, provided that the load and store are to different addresses. If either address is unknown, the conservative approach is to delay the load until both addresses are resolved. With memory forwarding, since the final address of a store is not known until the store actually completes, this delay would cause the conservative approach to never execute a load ahead of an earlier store.

Fortunately, there is a solution to this problem. A technique called *data dependence speculation* [12, 30] allows a load to speculatively execute before an earlier store, even if the store address is unknown. If it turns out that the load was not dependent on the store, then the speculation succeeds; otherwise, a true dependence has been violated, and the effects of the incorrect speculation must be undone. Recent out-of-order superscalar processors [19, 20, 24] have already implemented some form of data dependence speculation. With support for data dependence speculation, we can speculate that the final address of a reference will be the same as its initial address (i.e. we do not expect the reference to be forwarded), and therefore the delayed final-address generation will not degrade performance in the common case where the reference is not forwarded after all. If forwarding does occur, then our speculation would only be incorrect in the case where the load and store had different initial addresses but the same final address. In our experiments, we observed that incorrect data dependence speculation almost never occurred; hence it appears to be a very effective solution to supporting memory forwarding.

Handling Forwarding Cycles: A *forwarding cycle* is created when software erroneously inserts an address more than once into a forwarding chain. The hardware must have some mechanism for detecting and breaking forwarding cycles; otherwise, the machine could be stalled forever chasing the forwarding chain. Detecting forwarding cycles *accurately* is an expensive operation; for each hop, the hardware would have to match the current forwarding address against all previous forwarding addresses dereferenced by the same data reference. Because of this high cost—and also because we expect forwarding cycles to be extremely rare—we would prefer that the hardware instead perform a fast but possibly inaccurate check for a cycle during normal execution, and only perform accurate cycle detection when it is necessary. One possibility is to predetermine a limit on the number of forwarding hops that are allowed for a given data reference. We simply maintain a counter (which could be implemented either in hardware or in software) to keep track of the number of forwarding hops performed so far, and when this count exceeds the limit, we raise an *exception*. The corresponding software exception handler will then perform an accurate cycle check. If it is a false alarm, then we will reset the counter and resume execution; otherwise, the execution will be aborted.

Providing User-Level Traps Upon Forwarding: In addition to the *system-level* exception handlers which might be provided to support the dereferencing of forwarding addresses and the detection of forwarding cycles, it may also be useful to provide a lightweight *user-level* trapping mechanism that would be invoked upon accessing a forwarded location. Such a mechanism would be useful for allowing the application to tune its own performance in the following two ways. First, one could write a *profiling tool* to gather forwarding-related statistics for the purpose of improving the performance of a future execution of the program. For example, one might record which instructions experienced forwarding for the sake of eliminating that forwarding in future runs of the program. Second, a user-level trap handler could be used to optimize away forwarding (and thereby improve performance) *on-the-fly*. For example, one could write a tool that updates stray pointers on-the-fly to point directly to their correct final addresses, thereby avoiding the need to invoke the forwarding mechanism again. (Note that one must have application-specific knowledge in order to do this.) A mechanism similar to *informing memory operation* traps [17] could be used for this purpose.

3.3. Software Support

Having discussed the hardware support for memory forwarding, we now focus on its impact on *software*.

Initialization of Forwarding Bits: The forwarding bit of a memory word must already be clear when it is used by a program for the first time. To guarantee this, the operating system must perform an `Unforwarded_Write(0,0)` operation on all words in a region of memory to initialize it before making that memory available to an application.

Deallocating Forwarded Data: When an object is deallocated, all memory reachable via the chain of forwarding addresses for that object should be deallocated as well. A simple way to accomplish this is to create a *wrapper* memory-deallocation routine which first deallocates all of the memory allocated on the forwarding chain, and then calls the original memory-deallocation routine, which can be either a system-provided procedure (e.g., `free()` in C and `delete() in C++`) or a user-defined procedure if the program performs its own memory management.

Memory Alignment: Since the minimal granularity of memory forwarding is a word, software must ensure that two different objects which are being relocated to two different destinations do not share the same word, since we cannot store two different forwarding addresses in that same word. In other words, relocatable objects must be word-aligned. Enforcing this alignment can be accomplished either by specifying the alignment to the memory allocator for dynamically-allocated objects, or else by tuning the alignment option in the compiler if some relocatable objects are statically allocated. In some compilers—e.g., the MIPS C compiler that we used in our experiments—aggregate objects are already aligned to word boundaries by default.

Preserving Outcomes of Pointer Comparisons: The compiler is responsible for replacing all pointer comparisons that could be affected by relocation with explicit code to look up and compare final addresses. Pointer analysis techniques [14, 36] can help the compiler avoid inserting these more costly comparisons by ignoring cases where pointers cannot point to relocated objects.

4. Experimental Framework

To evaluate the potential performance benefits of memory forwarding, we modeled it in a modern processor and used it to enable a number of relocation-based optimizations which we applied to a collection of *non-numeric* applications. We chose to focus on non-numeric applications because compilers are mostly unable to guarantee the safety of data relocation in these applications. The goals of the optimizations that we applied were improving spatial locality and prefetching effectiveness. Since current compiler technology does not support these optimizations (mainly because their safety cannot be proven), we added these optimizations to the applications manually. Table 1 describes the eight applications used in our experiments along with the optimizations that we applied. All applications were run to completion in our simulations.

We added our proposed ISA extensions to the underlying MIPS ISA by making use of a few machine instruction sequences that never appear in ordinary programs (e.g., loading a value into a register which is hardwired to the value zero). We modeled the full performance effects of maintaining and dereferencing the forwarding addresses. The "*Space Overhead*" column shows the amount of virtual memory space for accommodating relocated data; this amount (ranging from 0.5MB to 14.9MB) does not present a problem in modern machines, and the simulation results include the impact of this overhead on performance.

We implemented data dependence speculation in our simulator. An ambiguous data dependence is stored in a table until

Table 1. Application characteristics. Note: "Inst. Grad." is the number of instructions actually graduated. The "combined" miss rate is the fraction of loads which suffer misses in both the 16KB D-cache *and* the 512KB L2 cache, using 32B cache lines. "Space Overhead" is the amount of virtual memory space used for forwarding addresses.

Name	Description	Source	Input Data Set	Optimizations Applied	Insts. Grad.	Load Miss Rate		Space Overhead
						D-Cache	Combined	
BH	Barnes-Hut's N-body force calculation algorithm	Olden [7]	4K bodies	Subtree clustering	1472M	2.57%	0.18%	1.7MB
Compress	Compresses and decompresses file in memory	SPEC95	A file of 150K characters	Array merging	546M	10.20%	0.46 %	0.5MB
Eqntott	Translation of boolean equations into truth tables	SPEC92	int_pri_3.eqn	Packing of hash table elements	1914M	5.22%	0.63%	0.5MB
Health	Simulation of the Columbian health care system	Olden	max. level = 5 max. time = 500	List linearization	213M	30.66%	16.73%	4.7MB
MST	Finds the minimum spanning tree of a graph	Olden	1K nodes	List linearization	302M	8.67%	5.35%	12.0MB
Radiosity	Virtual image rendering using hierarchical radiosity	IRISA [27]	A scene consisting of 10 lightly furnished rooms	List linearization	4552M	3.72%	0.26%	0.6MB
SMV	A symbolic model checker	CMU [26]	The "dme2.smv" file provided in the package	List linearization	302M	8.78%	3.75%	2.2MB
VIS	A verification and synthesis system for finite-state hardware systems	The VIS group [3]	A reduction of the 8 queens problem to combinational equivalence checking	List linearization	432M	12.81%	2.53%	14.9MB

Table 2. Simulation parameters.

Pipeline Parameters	
Issue Width	4
Functional Units	2 Integer, 2 FP, 2 Memory, 2 Branch
Reorder Buffer Size	64
Integer Multiply	3 cycles
Integer Divide	15 cycles
All Other Integer	1 cycle
FP Divide	15 cycles
FP Square Root	20 cycles
All Other FP	2 cycles
Branch Prediction Scheme	2-bit Counters

Memory Parameters	
Line Sizes	32/64/128/256/512 bytes
I-Cache	16KB, direct-mapped, 2 banks
D-Cache	16KB, direct-mapped, 2 banks
Data Victim Buffer	8 32-byte entries
Miss Handlers (MSHRs)	32 for data and 2 for inst.
Unified S-Cache	512KB, 2-way set-associative, 4 banks
Primary-to-Secondary Miss Latencies	12/18/30/54/102 cycles (plus any delays due to contention)
Primary-to-Memory Miss Latencies	75/93/129/201/345 cycles (plus any delays due to contention)
Primary-to-Secondary Bandwidth	16 bytes/cycle
Secondary-to-Memory Bandwidth	8 bytes/cycle

the two final data addresses involved in the dependence are determined. If the dependence is incorrectly speculated, then the simulator will then re-execute all instructions after (and including) the instruction which had violated the dependence. We replaced the memory deallocation calls in the applications by calls to our own memory deallocator which first checks whether there is any memory residing in forwarding chains which must be freed. We wrote a compiler pass in SUIF [35] that automatically determined which pointer comparisons needed to be replaced by final address comparisons. The overhead of executing these replaced comparisons is included in our simulations.

We performed detailed cycle-by-cycle simulations of our applications on a dynamically-scheduled, superscalar processor similar to the MIPS R10000 [38]. Our simulator models the rich details of the processor including the pipeline, register renaming, the reorder buffer, branch prediction, branching penalties, the memory hierarchy (including tag, bank, and bus contention), etc. Table 2 shows the parameters used in our model for the bulk of our experiments. Five line sizes—ranging from 32B to 512B—were used in our experiments, along with five corresponding sets of miss latencies (longer lines have longer transfer times).

We compiled our applications with `-O2` optimization using the standard MIPS C compilers and the SUIF compiler [35] under IRIX 5.3. For the experiments which required the insertion of software prefetches into the source code, we used the SUIF compiler; otherwise, the MIPS compiler was used.

5. Experimental Results

We now present results from our simulation studies. We start by evaluating the performance of a number of aggressive locality optimizations enabled by memory forwarding (which we simply refer to as *locality optimizations*). Next, we study the impact of these optimizations on prefetching effectiveness. We then examine the details of individual applications, explaining the optimizations that we applied to each application. Finally, we study the performance impact of forwarding overhead for one of the applications.

5.1. Performance of Locality Optimizations

Figure 5 shows the performance of our locality optimizations for various cache line sizes. Seven of our eight applications are included in Figure 5. We will show the performance of `SMV` separately, later in Section 5.4, since it is the only application that is affected by forwarding overhead. For each application in Figure 5, we show three line sizes, each of which has two cases: the bar on the left (**N**) is the original case where no locality optimization is applied, and the bar on the right (**L**) is the case with locality optimizations. For all applications except `BH`, the three line sizes used are **32B**, **64B**, and **128B**. For `BH`, we instead use line sizes of **32B**, **256B**, and **512B**, because the optimization applied to `BH` (subtree clustering) requires a cache line containing at least two tree nodes, and this requires cache lines longer than 128B (this optimization will not be turned on for lines shorter than 256B, and that is why the **N** and the **L** bars are identical for the **32B** line size in `BH`).

Each bar in Figure 5 represents execution time normalized to the **N** case of the **32B** line size, and is broken down into four cat-

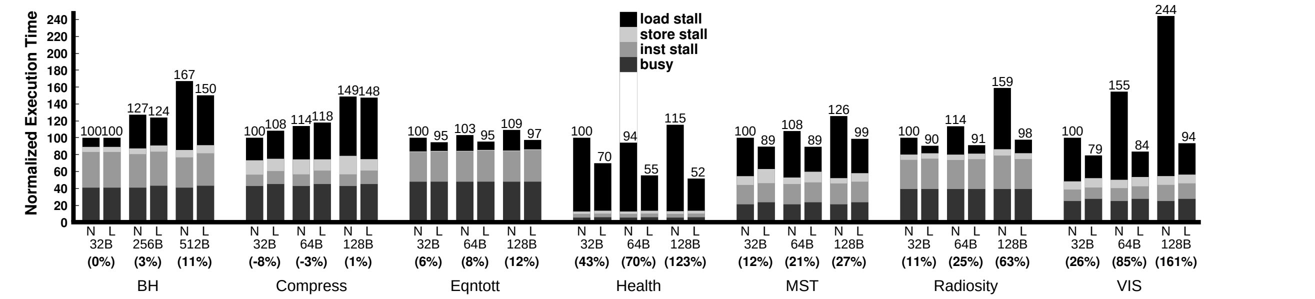

Figure 5. Performance of locality optimizations for various cache line sizes ($\mathbf{N}$ = not optimized, $\mathbf{L}$ = locality optimized).

egories explaining what happened during all potential graduation slots.[5] The bottom section (*busy*) is the number of slots when instructions actually graduate, the top two sections are any non-graduating slots that are immediately caused by the oldest instruction suffering either a load or store miss, and the *inst stall* section is all other slots where instructions do not graduate. Note that the *load stall* and *store stall* sections are only a first-order approximation of the performance loss due to cache stalls, since these delays also exacerbate subsequent data dependence stalls. In addition, there is a percentage in parentheses below each pair of bars representing the speedup of the optimized over the unoptimized case for the given line size.

Our first observation from Figure 5 is that performance generally degrades when line size increases, especially for the unoptimized cases. This trend is due to a lack of spatial locality in these applications, which means that longer lines offer little performance advantage. Fortunately, our locality optimizations (which are enabled by memory forwarding) improve the spatial locality of these application significantly. As we see in Figure 5, the optimized cases outperform the unoptimized cases for the same line sizes in all applications except `Compress`, and the speedups increase along with line size. The performance improvement can be dramatic—with 128B lines, `Health` and `VIS` enjoy more than twofold speedups. Among our optimizations, list linearization is particularly powerful since it improves the performance of `Health`, `MST`, `Radiosity`, and `VIS` substantially. It is interesting to note that in `Health`, the absolute performance of the optimized cases increases along with line size. This is due to the prefetching benefits of long cache lines after spatial locality is greatly improved. `Compress` is an exceptional case where the locality gets worse in the optimized case for 32B and 64B lines. We also observe from Figure 5 that the instruction overhead of these locality optimizations is usually low, which suggests that these optimizations could be invoked even more frequently during the execution to further improve the data layout.

While execution time is the most important performance metric, further insight can also be gained by examining the impact on total cache misses. Figure 6(a) shows the number of load D-cache misses in the unoptimized and optimized cases for different line sizes. Each bar is normalized to the **N** case of the **32B** line size, and is divided into two categories indicating how a D-cache miss is serviced. A *partial miss* is a D-cache miss that combines with an outstanding miss to the same line, and therefore does not necessarily suffer the full miss latency. A *full miss*, on the other hand, does not combine with any access and therefore suffers the full latency. Figure 6(a) clearly demonstrates that the improved spatial locality offered by locality optimizations reduces the miss count substantially, with more than a 35% reduction in misses in 11 out of the 21 cases (seven applications with three line sizes each). In many cases, both partial misses and full misses are reduced, and hence the total miss penalty decreases accordingly.

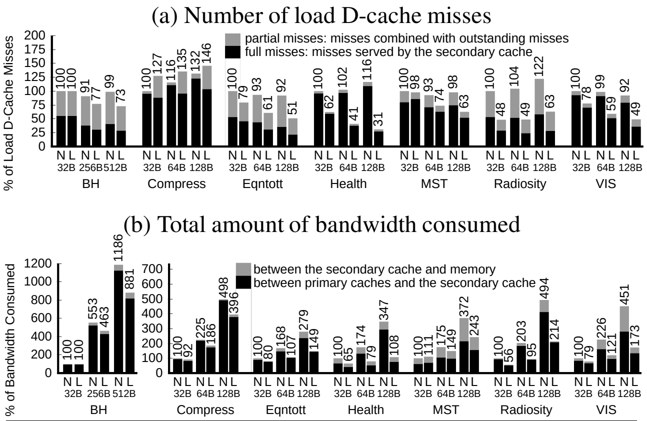

Figure 6. Additional performance metrics for the impact of locality optimizations ($\mathbf{N}$ = not optimized, $\mathbf{L}$ = locality optimized). The y-axes are normalized to the $\mathbf{N}$ cases of the $\mathbf{32B}$ line size.

Figure 6(b) shows another useful performance metric: the total amount of bandwidth consumed by our applications. Each bar in Figure 6(b) denotes the total number of bytes transferred between the primary and secondary caches (the bottom section), and the amount transferred between the secondary cache and main memory (the top section). Again, each bar is normalized to the **N** case of the **32B** line size. It is clear from Figure 6(b) that locality optimizations reduce the bandwidth consumption in nearly all cases, and achieve a bandwidth reduction of twofold or more in a few cases. Thus we see that these optimizations deliver not only higher performance, but also reduced bandwidth consumption.

5.2. Impact on the Effectiveness of Prefetching

We now turn our attention to the interaction between our locality optimizations and the effectiveness of prefetching. Based on a profile of each application, we added software prefetches for a few static loads that suffer significantly from cache misses. Prefetches are inserted at the earliest points in the program where the prefetch addresses are known (this is done in an identical fashion for both the original and locality optimized cases). We assume

[5] The number of graduation slots is the issue width (4 in this case) multiplied by the number of cycles. We focus on graduation rather than issue slots to avoid counting speculative operations that are squashed.

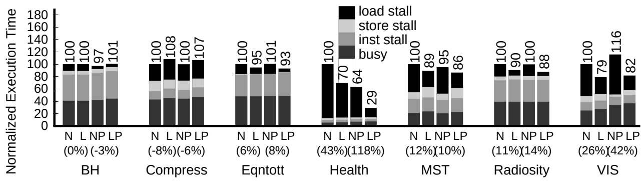

Figure 7. Performance impact of locality optimizations on prefetching. (N = not optimized, L = locality optimized, NP = prefetching without locality optimizations, LP = prefetching with locality optimizations).

that a single prefetch instruction can prefetch one or more consecutive cache lines (i.e. *block prefetching* is supported). For both the unoptimized and optimized cases, we experimented with a range of prefetch block sizes, and we report the results with the block size that performed the best for each case.

Figure 7 shows how prefetching performs both with (**LP**) and without (**NP**) locality optimizations. For the sake of comparison, the **N** and **L** cases from Figure 5 are also included in Figure 7. The cache line size is fixed at 32B. We observe from Figure 7 that the performance of prefetching is improved by locality optimizations in five applications, and two of them (`VIS` and `Health`) enjoy speedups of over 40%. We note that four of these five applications operate heavily on linked lists, and previous research [25] has shown that prefetching linked lists—especially those that are short and traversed within small loop bodies—is particularly difficult because of the pointer-chasing problem. As we can see in Figure 7, the *list linearization* optimization is quite successful in alleviating this problem. With the exception of `VIS` (which experiences considerable prefetching overhead), in the remaining four out of five applications where the locality is substantially improved, combining locality optimizations and prefetching (**LP**) performs better than either technique alone (most noticeably in `Health`). Therefore, it appears that prefetching and our locality optimizations are complementary in nature.

5.3. Case Studies

Having studied the overall performance, we now look at the individual applications in more detail.

Health, MST, Radiosity, and VIS: We apply the same locality optimization to all four of these applications: *list linearization*. The structure of the linked lists used in these applications is modified throughout the program execution, and therefore list linearization is invoked periodically. To make our discussion more concrete, we use `VIS` as a representative example. `VIS` is a large application, consisting of more than 150,000 lines of C code. This program makes extensive use of a generic list library which implements many common list operations. Our optimizations are localized within this library. We optimize the locality of list processing as follows. We add a counter field to the head record of each list to count how many insertion or deletion operations have been performed on the list since the last time that the list was linearized. The list linearization procedure `ListLinearize()`—shown earlier in Figure 4(b)—is invoked whenever the list's counter exceeds a threshold, which was arbitrarily set to 50 in our experiments. The counter is reset after each linearization. Despite the simplicity and usefulness of this optimization, performing it without the support of memory forwarding is dangerous due to the fact

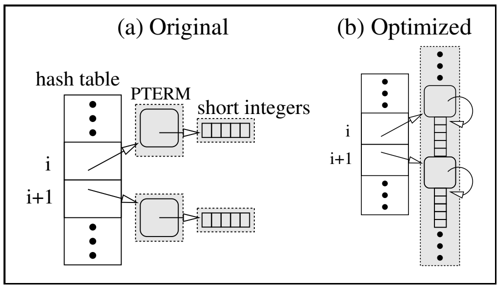

Figure 8. Locality optimization for `Eqntott` (objects in the same shaded region are allocated to contiguous memory).

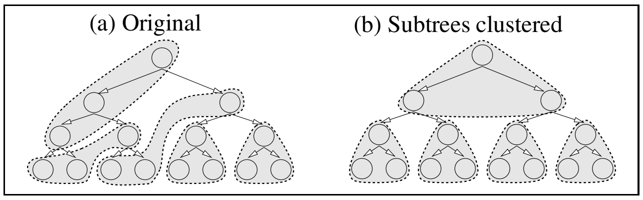

Figure 9. Example of the subtree clustering applied to `BH` (nodes in the same shaded region are in the same cache line).

that most functions in this library return pointers to list elements, which can be scattered across any of the over hundred source files of `VIS`. The program behave incorrectly if after a list is linearized, it is later accessed using a pointer to the middle of the list that existed before the linearization. Fortunately, memory forwarding allows us to simply ignore this hazard, thereby safely resulting in an over twofold performance gain with 128B lines.

Eqntott: The most interesting data structure in `Eqntott` is a hash table which stores pointers to a record of type `PTERM`. A `PTERM` record in turn contains a pointer to an array of short integers. The original layout of this data structure is shown in Figure 8(a). We optimize the locality by (i) relocating a `PTERM` record and its short integer array into a single chunk of memory, and (ii) putting these chunks into contiguous memory locations in increasing order of the hash index. The optimized layout is shown in Figure 8(b). This relocation optimization is invoked only once in the program, immediately after the hash table is constructed.

BH: In `BH`, an octree is constructed and then traversed at each time step of the N-body force calculation. The octree is constructed in a depth-first order, but the traversal order is fairly random. We improve the locality of the traversal by clustering non-leaf nodes of the tree. We do not cluster leaf nodes since they are actually linked together by a list and accessed via list traversals. Subtree clustering [11] attempts to pack nodes of a subtree into a cache line, in the most balanced form. Locality will be improved if the next node to be visited—which can be any of the children of the current node—is already in the current cache line. Figure 9 illustrates this optimization using a binary tree. Figure 9(a) shows the original memory layout of the tree, which was created using a pre-order traversal, and Figure 9(b) shows the memory layout after subtree clustering. Since a non-leaf node in `BH` is 78B long, we need cache lines of 256B or longer to do meaningful clustering.

Compress: The most relevant data structures in `Compress` are two hash tables, namely `htab` and `codetab`, which are implemented using arrays. Indices to `htab` are computed through

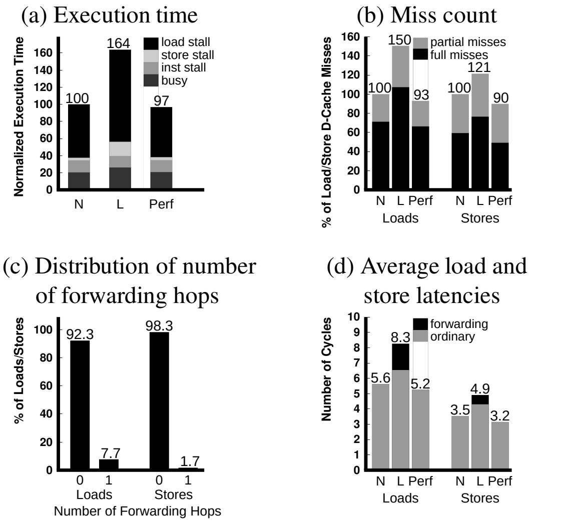

Figure 10. Performance results for SMV. (N = not optimized, L = locality optimized with realistic forwarding, Perf = locality optimized with perfect forwarding). The line size is fixed at 64B.

hashing, but `codetab` always shares the same index values as `htab`. Therefore, spatial locality might be improved if `codetab[i]` could be next to `htab[i]` in the memory. We achieve this by copying the two tables into a single larger table `T` such that `htab[i]` and `codetab[i]` occupy adjacent elements in `T`. However, as we have already seen in Figure 5, performance is in fact degraded by this optimization for 32B and 64B lines due to worse locality than in the original code.

5.4. Impact of Forwarding Overhead

In each of the applications that we have studied so far, we were successful enough at updating the appropriate pointers to point to a relocated object's new location that the forwarding mechanism was almost never invoked. (At the same time, we would like to point out that without memory forwarding support, we would not have been able to apply these optimizations because they were not provably safe.) As a result, the performance of dereferencing a forwarding address did not matter in these cases. To quantify the impact of forwarding overhead in a case where it does matter, we now focus on SMV, which is the only application we studied that experiences significant forwarding after data relocation.

SMV is a model checking program which makes extensive use of Binary Decision Diagrams (BDDs) [4]. The BDD nodes are connected both through a hash table and through binary trees. The hash table is organized as an array of buckets pointing to linked lists. Since more cache misses occur during hash table accesses than binary tree accesses, we attempted to improve locality by linearizing the lists stored in the hash table. Unfortunately, since our optimized code is not able to update the tree pointers to point to a relocated object's new address, forwarding does occur whenever relocated BDD nodes are accessed via the tree pointers.

Figure 10 shows our performance results for SMV. In addition to the cases without (**N**) and with (**L**) locality optimization, as shown in earlier graphs, we also show a case with locality optimization and *perfect* forwarding (**Perf**). We say that memory forwarding is perfect if all references to relocated objects access them directly at their new addresses, and hence no forwarding is actually required. While this latter case is not achievable, it represents a useful bound on performance. As we see in Figure 10(a), the performance of scheme **L** is degraded by forwarding in two ways. First, the act of dereferencing forwarding addresses incurs extra time. Second, when forwarding occurs, both the old and new locations of relocated data are accessed, thereby degrading cache behavior. With perfect forwarding, there is no forwarding overhead and the performance does improve. However, the improvement is only marginal due to the fact that we cannot optimize the layout to accelerate both the hash table and tree access patterns.

To provide further insight into the source of the forwarding overhead, Figure 10 presents three additional performance metrics. Figure 10(b) shows the impact of the schemes on the number of load and store data cache misses. As we see in this figure, scheme **L** suffers a noticeable increase in misses. Figure 10(c) shows that 7.7% of loads and 1.7% of stores require one forwarding hop under scheme **L**. Finally, Figure 10(d) shows the average number of CPU cycles needed to complete a load or store under each scheme. Each bar in Figure 10(d) is divided into two sections explaining the reason for the stall. The *forwarding* section represents the time spent dereferencing forwarding addresses, and the *ordinary* section includes cache hit and miss latencies. The *ordinary* sections of scheme **L** increase due to the cache pollution effects of touching the forwarding pointers, as mentioned earlier. As we see in Figure 10(d), both the latency of dereferencing a forwarding address and its resulting cache pollution effects play significant roles in the overall performance degradation. A profiling tool based on user-level traps (as discussed earlier in Section 3.2) could potentially identify cases such as this where forwarding occurs too frequently.

6. Conclusions

As changes in technology continue to alter the landscape of what constitutes a major performance bottleneck, it is sometimes worth re-examining old architectural ideas that have fallen out of fashion to see whether they can be adapted to serve completely new purposes. In this paper, we have examined such a technique: *memory forwarding*. Although the original concept was proposed to facilitate garbage collection in early Lisp machines, we have demonstrated that memory forwarding can be adapted to address the entirely modern problem of enhancing cache performance. In addition, we have shown that it is quite feasible to implement this mechanism within modern out-of-order superscalar processors, largely because forwarding can be treated as an exception.

By liberating aggressive relocation-based data layout optimizations from concerns over violating program correctness, memory forwarding can enable impressive performance gains: we observe over twofold speedups in two applications. These optimizations are useful not only for hiding memory latency, but also for reducing memory bandwidth consumption. Although one must still exercise caution not to use forwarding carelessly, a user-level trap mechanism can help identify and avoid cases where pointers have not been updated successfully. In summary, memory forwarding is a powerful tool which makes a large class of optimizations that were promising in theory useful in practice. Its applicability extends beyond caches to the rest of the memory hierarchy (e.g., disks), and we advocate that it be supported in future processors.

7. Acknowledgments

We thank Daniel Meneveaux for providing his radiosity program. Chi-Keung Luk is partially supported by a Canadian Commonwealth Fellowship. Todd C. Mowry is partially supported by an Alfred P. Sloan Research Fellowship, and by a Faculty Development Award from IBM.

References

[1] H. G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, April 1978.

[2] D. G. Bobrow and D. W. Clark. Compact encodings of list structure. *ACM TOPLAS*, 1(2):267–286, October 1979.

[3] R. K. Brayton, G. D. Hachtel, A. S. Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. R. Shilpe, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification*, July 1996.

[4] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, C-35(8):677–691, 1986.

[5] D. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *ISCA'96*, pages 78–89, May 1996.

[6] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *ASPLOS-VIII*, October 1998.

[7] M. C. Carlisle and Anne Rogers. Software caching and computation migration in olden. In *Proceedings of PPoPP'95*, pages 29–38, July 1995.

[8] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *ASPLOS-VI*, pages 252–262, October 1994.

[9] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Trans. on Comp.*, 44(5), May 1995.

[10] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, Nov 1970.

[11] T. M. Chilimbi, J. R. Larus, and M. D. Hill. Tools for cache-conscious data structures. In *PLDI'99*, May 1999.

[12] G. Chrysos and J. Emer. Memory dependency prediction using store sets. In *ISCA'98*, pages 142–153, June 1998.

[13] D. W. Clark. *List structure: measurements, algorithms, and encodings*. PhD thesis, Carnegie-Mellon University, August 1976.

[14] R. Ghiya and L. J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[15] R. Greenblatt. The LISP Machine. Technical Report Working Paper 79, M.I.T. Artificial Intelligence Laboratory, November 1974.

[16] W. J. Hansen. Compact list representation: Definition, garbage collection, and system implementation. *Commun. ACM*, 12(9):499–507, September 1969.

[17] M. Horowitz, M. Martonosi, T. C. Mowry, and M. D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *ISCA'96*, pages 260–270, May 1996.

[18] A.S. Huang and J. P. Shen. A limit study of memory requirements using value reuse profiles. In *MICRO-28*, pages 71–81, Dec 1995.

[19] D. Hunt. Advanced performance features of the 64-bit PA-8000. In *IEEE CompCon'95*, March 1995.

[20] IBM. *PowerPC 620 Risc Microprocessor Technical Summary*, October 1994.

[21] T.E. Jeremiassen and S.J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of PPoPP'95*, July 1995.

[22] R. E. Kessler and M. D. Hill. Page placement algorithms for large real-indexed caches. *ACM Trans. on Comp. Sys.*, 10(4), Nov 1992.

[23] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV*, pages 63–74, April 1991.

[24] D. Leibholz and R. Razdan. The Alpha 21264: A 500 MHz Out-of-Order Execution Microprocessor. In *IEEE CompCon'97*, February 1997.

[25] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *ASPLOS-VII*, pages 222–233, October 1996.

[26] K. L. McMillan. *The SMV system*. Carnegie-Mellon University, Feb 1992.

[27] D. Meneveaux, K. Bouatouch, and E. Maisel. Memory management schemes for radiosity computation in complex environment. Technical Report PI 1097, IRISA/INRIA, 1996.

[28] M. L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, M.I.T., Cambridge, Mass., 1963.

[29] D. A. Moon. Architecture of the symbolics 3600. In *ISCA'85*, pages 76–83, 1985.

[30] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA'97*, pages 181–193, June 1997.

[31] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V*, pages 62–73, October 1992.

[32] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn. Evaluation of the SPUR Lisp architecture. In *ISCA'86*, pages 444–452, 1986.

[33] O. Temam, E. D. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *Proceedings of Supercomputing'93*, pages 410–419, November 1993.

[34] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. on Comp.*, 43(6):651–663, 1994.

[35] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W.K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12), Dec 1994.

[36] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI'95*, pages 1–12, June 1995.

[37] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI'91*, pages 30–44, June 1991.

[38] K. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.